1

EECS 280 – Lecture 15

Deep Copies and the Big Three

C++

2

Destructors and Polymorphism

```
class Player {
public:
  ...
  ~Player() {}
  ...
};

class SimplePlayer : public Player {
  vector<Card> hand;
public:
  ...
  ~SimplePlayer() {}
  ...
};

int main() {
  Player *player = new SimplePlayer(...);
  ... // do stuff with player
  delete player;
}
```

Because the destructor is non-virtual, we end up using this one!

The hand vector does not get destructed.

没有被删除

This will break.

Static type here is Player.

5/31/2022

① subtype polymorphism 中的 dtor 要是 virtual.

3

Destructors and Polymorphism

```
class Player {
public:
  ...
  virtual ~Player() {}
  ...
};

class SimplePlayer : public Player {
  vector<Card> hand;
public:
  ...
  virtual ~SimplePlayer() {}
  ...
};

int main() {
  Player *player = new SimplePlayer(...);
  ... // do stuff with player
  delete player;
}
```

即使base class中的destructor是空的，我们也要将其变成virtual的形式，因为这样在程序run time 的时候，就能根据dynamic type来分配正确的destructor，确保数据都被清理干净。

If there are subtypes, ALWAYS make the destructor virtual.

Now use this one! It implicitly calls the destructor for hand.

This will work.

Dynamic type here is SimplePlayer.

5/31/2022

① 即使 dtor 为空，依然 virtual.

4

Recall: Copying Compound Objects

- The built-in behavior for copying compound objects is just a straightforward member-by-member copy.

```
class Example {
private:
  int x; double y;
public:
  Example(int x_in,
          double y_in)
    : x(x_in), y(y_in) { }
};

int main() {
  Example e1(2, 3.5);

  // init e2 as copy of e1
  Example e2(e1);

  // This syntax is equivalent
  // Example e2 = e1;
}
```

The Stack: main — e1 Example: 0x1000 2 x, 0x1004 3.5 y; e2 Example: 0x1012 2 x, 0x1016 3.5 y

我们可以使用这中内置的方式，直接复制class中的所有属性。

5/31/2022

① class-type object 的内置 copy 方式

② $e_1(e_2)$ 和 $e_1 = e_2$

6

Two Kinds of Copies

- Two contexts in which we can copy the value from one object to another:
  1. Initialization (also used for parameter passing)
  2. assignment

Initialization is giving a first-time value as part of creating the object.

```
int func(Example a); // pass by value
int main() {
  Example a1(2, 3.5);
  Example a2(a1);  // init a2 as copy of a1
  Example a3 = a1; // init a3 as copy of a1
  func(a1);        // init parameter a as copy of a1
}
```

5/31/2022

① 两种不同的 copy.

② initialization 的判断条件.

③ initialization 的两种类型.

7

The Copy Constructor

- When a compound object is **initialized** from another of the same type, a **copy constructor** is used.

```
class Example {
private:
  int x; double y;
};

int main() {
  Example e1(2, 3.5);

  // init e2 as copy of e1
  Example e2(e1);
}
```

Question

Which of these is a valid copy ctor for Example?

```
class Example {
public:
  A Example(int x_in, double y_in)
      : x(x_in), y(y_in) { }
  B Example(const Example &other)
      : x(other.x), y(other.y) { }
  C Example(const Example &other)
      : Example(other) { }
  D Example(Example other)
      : x(other.x), y(other.y) { }
  E Example(const Example &other) {
      x = other.x;
      y = other.y;
    }
};
```

by value copy ctor

① "=" 和 "()" 在 initialization 时是一样的，都 call copy constructor

② assignment 时另能 "=" 不能 "()".

8

The Copy Constructor

- Why is the parameter for the copy constructor passed by reference?

```
class Example {
public:
  Example(const Example &other);
};
```

YES

- What would happen if we used pass-by-value instead?

```
class Example {
public:
  Example(Example other);
};
```

NO

在function 的parameter的位置上pass by value相当于调用了copy constructor，但是我们这个congstructor的意义就是为了实现copy的功能，所以就会陷入一个无限循环当中

5/31/2022

① 为何 Copy-constructor 的 parameter 要用 pass-by-reference.

9

Implicitly Defined Copy Constructor

- In the previous exercise, we wrote out a copy constructor for the `Example` class:

```
class Example {
public:
  Example(const Example &other)
    : x(other.x), y(other.y) { }
};
```

- **HOWEVER**, all classes[1] **already** have a built-in copy constructor just like this that the compiler provides for you.
  - It just does a **member-by-member** copy.
  - The built-in version is provided even if you made other constructors (unlike the built-in default ctor).

与destructor不同的是，如果你自己定义了一个destructor的function，那么compiler将不会在调用default的constructor，但是对于copy constructor来时，它要常见，所以即使你自己定义了一个copy constructor，compiler也依然会调用内置的default的copyconstructor

1 And structs & unions, too.

5/31/2022

① build in copy constructor 的工作原理.

② 自主定义的 Copy Constructor 并不影响 build-in 的 Copy constructor.

10

UnsortedSet Copy Constructor

- Here's the built-in copy constructor for `UnsortedSet`:

```
template <typename T>
class UnsortedSet {
private:
  T *elts;
  int capacity;
  int elts_size;
public:
  UnsortedSet(const UnsortedSet &other)
    : elts(other.elts),
      capacity(other.capacity),
      elts_size(other.elts_size) { }
};
```

Compiler Provides

- Is this the copying behavior we would want for `UnsortedSet`?

① build-in copy constructor 在 unsorted set 中的应用.

(产生错误)

11

The Shallow Copy Problem

- The built-in **shallow** copying behavior doesn't work well with objects used **indirectly**.
  - It doesn't follow pointers!
  - It just copies the address.

UnSet s2(s1);

The Stack: main — s2 UnsortedIntSet: 0x1016 0x80 elts, 0x1024 0 elts_size, 0x1028 0 capacity; s1 UnsortedIntSet: 0x1000 0x9992 elts, 0x1008 2 elts_size, 0x1012 2 capacity; The Heap: 2 3

The Stack: main — s2 UnsortedIntSet: 0x1016 0x9992 elts, 0x1024 2 elts_size, 0x1028 2 capacity; s1 UnsortedIntSet: 0x1000 0x9992 elts, 0x1008 2 elts_size, 0x1012 2 capacity; The Heap: 5 10

5/31/2022

① 对于使用 indirectly 的 object，shallow copy 会产生的问题

12

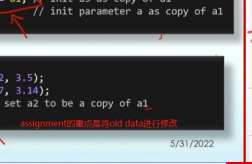
Shallow Copies vs. Deep Copies

13

Exercise: Deep Copy

- Write a custom version of the `UnsortedSet` copy constructor that implements a **deep copy**.

```
template <typename T>
class UnsortedSet {
public:
  UnsortedSet(const UnsortedSet &other)
  {
  }
};
```



See file L15.1_UnsortedIntSet on Lobster.

5/31/2022

14

Solution: Custom Copy Constructor

- We need a **deep copy** for `UnsortedSet`.

```
template <typename T>
class UnsortedSet {
public:
  UnsortedSet(const UnsortedSet &other)
    : elts(new T[other.capacity]),
      capacity(other.capacity),
      elts_size(other.elts_size) {

    for (int i = 0; i < elts_size; ++i) {
      elts[i] = other.elts[i];
    }
  }
};
```

Allocate our own array with same size as the other.

Copy over each element individually.

It's ok to access other.elts, even though it's private. We're still in the UnsortedSet scope!

① Custom copy constructor (deep) implementation

14

Solution: Custom Copy Constructor

这是我们自己定义的 copy constructor，可以做到deep copy 所以compiler将不会call default 的 copy constructor，因为default的是shadow copy

- We need a **deep copy** for `UnsortedSet`.

```
template <typename T>
class UnsortedSet {
public:
  UnsortedSet(const UnsortedSet &other)
    : elts(new T[other.capacity]),
      capacity(other.capacity),
      elts_size(other.elts_size) {

    for (int i = 0; i < elts_size; ++i) {
      elts[i] = other.elts[i];
    }
  }
};
```

Allocate our own array with same size as the other.

Copy over each element individually.

It's ok to access other.elts, even though it's private. We're still in the UnsortedSet scope!

18

Two Kinds of Copies

- Two contexts in which we can copy the value from one object to another:
  - Initialization (also used for parameters)
  - Assignment

Initialization is giving a value as part of creating the object.

```
int func(Example a); // pass by value
int main() {
  Example a1(2, 3.5);
  Example a2(a1);  // init a2 as copy of a1
  Example a3 = a1; // init a3 as copy of a1
  func(a1);        // init parameter a as copy of a1
}
```

initialization是由constructor来管

Assignment is giving a new value to an object that already exists.

```
int main() {
  Example a1(2, 3.5);
  Example a2(7, 3.14);
  a2 = a1; // set a2 to be a copy of a1
}
```

assignment是由operator来管

assignment的重点是将old data进行修改

5/31/2022

① initialization 是由化来实现.

② assignment 是由开化来实现.

19

The Assignment Operator

- **Assigning** the value of one compound object to another uses an overloaded **assignment operator**.

```
class Example {
public:
  Example & operator=(const Example &rhs);
};
```

- If you don't provide your own assignment operator, the compiler implicitly makes one for you.
  - It just does a **member-by-member** copy.
  - For `Example`, it would look something like this...

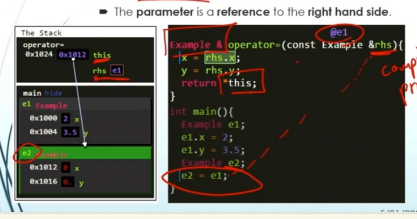
```
class Example {
public:
  Example & operator=(const Example &rhs) {
    x = rhs.x;
    y = rhs.y;
    return *this;
  }
};
```

① 内置 default assignment operator 的工作原理.

20

The Assignment Operator

- Overloads for the assignment operator must be implemented as **member functions**.
- When the function is called...
  - The **this pointer** points to the **left hand side**.
  - The **parameter** is a **reference** to the **right hand side**.

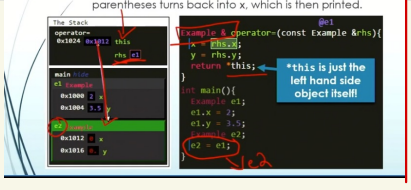


① overload "=" 一定要 return class type. 因为是 member function. 所以 "this" 是 "=" 之前 object 的 pointer. "*this" 是 class type.

21

return *this

- Semantically, an assignment expression is supposed to evaluate back into its left hand side (the object assigned to).
- Example: `cout << (x = 3) << endl;`
- We first assign 3 into x, but then the whole thing in parentheses turns back into x, which is then printed.



x = y = 10;

set1 += 3 += 5;

① "=" 结束是左侧 variable 本身.

22

Copy Ctor vs. Assignment Operator

initialization

replace/overwrite

```
UnSet s2 (s1);
UnSet s2 = s1;
UnSet s3; // value
UnSet s4; // value
s4 = s3;
```

对于assignment来说，就要重新指向一个新的地址，这个地址复制了s3的set array中的值，再让s4指向新的地址，之后free旧的地址

5/31/2022

23

Custom Assignment Operator

- We need a **deep copy** for `UnsortedSet`.

```
template <typename T>
class UnsortedSet {
public:
  UnsortedSet & operator=(const UnsortedSet &rhs) {
    if (this == &rhs) { return *this; }

    delete[] elts;
    elts = new T[rhs.capacity];
    capacity = rhs.capacity;
    elts_size = rhs.elts_size;

    for (int i = 0; i < elts_size; ++i) {
      elts[i] = rhs.elts[i];
    }

    return *this;
  }
};
```

Check for self-assignment.

Kill old array.

Make new array.

Copy over all the elements.

Return the LHS object.

若它们两个 class 其实是一个 class.

① 在 custom assignment operator "=" 定义时的步骤. ① 都是你自己 ts2 = ts1

② 删旧 arr

③ 创新 arr

④ 遍历新 arr

⑤ 返回新 object.

24

Recap: Deep Copies

- Sometimes we need to provide custom implementations for the **copy constructor** and **assignment operator**.
  - The built-in behavior gives us a **shallow copy**, and that is very bad.
- `UnsortedSet` is an example of this.
  - In this case, a proper copy should be **deep**.
  - A copy should have its-own dynamically allocated array – not just a pointer to someone else's!

① custome copy construct

② custome assignment operator

③ UnsortedSet 是个需要 ①和② 的例子.

26

The Rule of The Big Three

- The Big Three
  - Destructor
  - Copy Constructor
  - Assignment Operator
- The rule of the big three is:
  - If you need to provide a **custom** implementation for **any** of them...

  ...you almost certainly need to provide a custom implementation for **all** of them.

① Big three 所指的具体内容

② Big three 用优意则全部用.

27

Implicitly Defined Big Three

- All objects have the big three.
  - If you don't provide a custom version, the compiler provides **implicitly defined** versions for you.
- Implicit destructor
  - No special cleanup
- Implicit copy constructor
  - Shallow copy
- Implicit assignment operator
  - Shallow copy

当你需要删除一个class的时候，也要删除class里面带的一个指向heap的pointer所指的array，这和copy的道理是一样的

5/31/2022

① 系统自带的 big three 的实现原理.

28

Custom Big Three

- When do we need our own **custom** versions?
  - If you need a deep copy.
  - You need a deep copy if the object owns and manages any resources (e.g. dynamic memory).
- Hints:
  - Check the constructor. If it creates dynamic memory, you probably need the big three.
  - Look at the members. If some of them are pointers, you might need the big three.

① 如果用到 dynamic memory 可能会用 big three.

② 如果 member variable 有 Pointer 可能会用到 big three.

29

The Big Three

- Destructor
  1. Free resources[1]
- Copy Constructor
  1. Copy regular members from `other`
  2. Deep copy resources from `other`
- Assignment Operator
  1. Check for self-assignment
  2. Free old resources
  3. Copy regular members from `rhs`
  4. Deep copy resources from `rhs`
  5. return `*this`

1 The "resource" we often see in 280 is dynamic memory.

5/31/2022

① Big three 具体功能.